AD-A274 391

AFIT/GCE/ENG/93D-11

AN OBJECT-ORIENTED DATABASE INTERFACE FOR ADA

THESIS

Anthony David Moyers
Captain, USAF

AFIT/GCE/ENG/93D-11

DTIC ELECTE DEC 27 1993 S E D

93-31038

Approved for public release; distribution unlimited

93 12 22 151

AFIT/GCE/ENG/93D-11

AN OBJECT-ORIENTED DATABASE INTERFACE FOR ADA

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Anthony David Moyers, B.S.

Captain, USAF

December, 1993

Accesion For		
NTIS CRA&I	✓	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

DTIC QUALITY INSPECTED 2

Approved for public release; distribution unlimited

Acknowledgements

I would like to thank my thesis advisor, Maj Mark Roth, for his patience and guidance over the past year. I also wish to thank fellow students, Capt Karl Mathias for his helpful suggestions and optimistic nature, and Capt Tim Halloran for his technical assistance. Finally, I would like to thank my friends Hal, Rebecca, Bill, Martin, and "Buffy" who encouraged and supported me while keeping my phone bills at an all time high.

Anthony David Moyers

Table of Contents

List of Figures

List of Tables

AFIT/GCE/ENG/93D-11

Abstract

Object data management (ODM) is currently at the forefront of research and development efforts in the database community. This exciting new field has emerged as an answer to increasingly complex data management problems. Many promising prototypes and commercial products are emerging. In order to take advantage of ODM technology, the Department of Defense (DoD) needs to ensure that these systems are accessible to Ada programmers. Ada is the standard programming language for the DoD and is therefore used in many important defense-related software development efforts.

The Air Force Institute of Technology (AFIT) is conducting research towards the development of an Ada ODM interface to existing ODM systems. The design goals are portability, transparency, and completeness for Ada programmers. Portability means that the ODM system can be changed without affecting any existing application programs. Transparency means that Ada programmers can use the interface without having to know different programming languages or specifics about the different systems being used. Completeness means that all of the functionality of the ODM systems are available to Ada application developers.

This thesis defines requirements for an Ada ODM interface and proposes a design. In addition, the challenges associated with implementation are investigated using commercial ODM systems at AFIT. Implementation of the interface is based on the use of Ada bindings to the existing application program interfaces (APIs) of the ODM systems. A preprocessor will be necessary in order to achieve transparency.

AN OBJECT-ORIENTED DATABASE INTERFACE FOR ADA

I. Introduction

Object data management (ODM) is currently at the forefront of research and development efforts in the database community. This exciting new field has emerged as an answer to increasingly complex data management problems. Many promising prototypes and commercial products are emerging. In order to take advantage of ODM technology, the Department of Defense (DoD) needs to ensure that these systems are accessible to Ada programmers. Ada is the standard programming language for the DoD and is therefore used in many important defense-related software development efforts. This research effort defines requirements for an Ada interface, recommends a design solution, and looks at the associated implementation challenges.

1.1 Background

The management of information has become essential to the daily work of industry and government. For this reason, database management systems (DBMS) have become a central component in modern computing. The DBMS provides a convenient and efficient environment for the creators of computer applications to retrieve, store, and manipulate large bodies of data. DBMS responsibilities include integrity control, query processing, recovery, concurrency control, transaction processing, data security and more.

The use of the DBMS has led to the development of many different data models to describe data, data relationships, data semantics, and consistency constraints. DBMS technology has evolved over the last thirty years in three generations which can be distinguished according to logical data model. Originally, data was stored in flat file systems and all of the functionality mentioned above was left up to the applications. First generation DBMSs are based on the network and hierarchical models. These models require data to be stored in fixed-format records of several types. The records define a fixed number of fields and each field is a fixed length. In the network and hierarchical models, the relationships between records are represented by links or pointers. The network model is arranged as a collection of arbitrary graphs, while the hierarchical model is arranged as a collection of trees. The second generation of DBMS is based on the relational model which represents data and data relationships by a collection of tables, each of which has a number of columns with unique names. While also a fixed-format record based model, the relational model relates records by the values they contain rather than pointers. This feature allows a formal mathematical foundation to be defined for the relational model. For this reason and many others, the relational database has become the most popular and widely used data model.

Despite its popularity, the relational model has many shortcomings for representing and describing information. These deficiencies are major problems for DBMS developers who want to support new kinds of applications which have requirements much different than traditional business applications. Traditional database applications manipulate large numbers of similar data items or records which are short and fixed length. These applications are characterized by short transactions requiring no human interaction. The database schema is changed infrequently and usually only simple changes are required.

Newer applications have much different requirements including the manipulation of complex, interrelated data for science and engineering purposes. Examples include computer-aided design (CAD), computer-aided software engineering (CASE), multimedia, and expert systems. Advances in computer hardware and database technology have made these types of applications practical candidates for database management. Among the data elements introduced by the new applications are complex objects, behavioral data, rules information, and media information. This data requires new data models, new transaction models, and new query facilities.

These new requirements have led to the developement of the third generation of DBMS. Third generation systems are not so clearly defined. They have become known as object data management systems (ODMSs) because they all profess to store and manipulate objects in the sense of the object-oriented programming paradigm. The current state of ODM is characterized by the lack of a common data model and the lack of formal foundations. Many prototypes and products are emerging but with no standard design methodology. There is still much debate about what approach should be taken to implement ODM concepts. Several approaches have been proposed, but the extended object-oriented programming language (OOPL) and the extended relational database management system (RDBMS) are emerging as the most popular and the most promising. The extended OOPL implements an ODMS by extending an existing programming language with DBMS facilities. This approach has been very popular in research prototypes and commercial products. The second approach implements ODMS concepts by extending the capabilities of a relational query language. Extended RDBMSs retain the benefits of many years of experience and research with the relational model (6).

Table 1.1 provides a summary of engineering design tool requirements and shows how these requirements are supported in second and third generation DBMSs (10). Column one contains au ODM requirement, while column two states how the requirement could be realized in an engineering design system. Columns three and four explain how the requirement is supported in the traditional DBMS versus the object-oriented DBMS.

The differing approaches to ODM and the state of flux in ODM technology have led to some confusion over terminology. Cattell (6), for example, uses the term ODM to include systems based on object-oriented, semantic, functional, and extended relational models of data. In this thesis, the term ODM system refers to systems that use the object-oriented data model and are characterized by integration with existing programming languages.

1.2 Problem Statement

One of the most important components of a DBMS regardless of its data model is the Application Program Interface (API). The API is the tool through which a user's programs interact with the DBMS. The usefulness of a DBMS, to a large extent, depends on the number and type of programming languages supported by the API. Many of the ODM systems currently available support APIs using the C or C++ programming language. These languages are widely used in industry and the decision to support them is driven by supply and demand.

Characteristic	Design Tool Example	Traditional DBMS	Object-Oriented DBMS
Complex State	References to subcomponents within circuit.	A key is required for each sub-component. Joins are required to merge into a single object.	Fundamental to the object-oriented paradigm.
Inheritance	New adder inherits attributes of a typical adder and modifies them to fit a particular circuit.	Complete specification of the schema must be defined *a priori*.	Fundamental to the object-oriented paradigm.
Complex Data Types	Graphical representation of a circuit.	Only supports basic data types such as integer and character.	Supports graphical and textual data and allows user to define data types.
Multiple Views	Top level view of design or more detailed look at a sub-component.	Must be defined in the application. Limited by record oriented retrieval.	Can be specified as a method for the object. Data is more easily retrieved using object-oriented storage techniques.
Multiple Versions	Current and historical versions.	May support multiple versions of individual records.	Generally built in as a tree structure with root node representing a version. Tree includes all objects which make up the version.
Phased Development	Top down design.	Not supported. Entire schema must be defined *a priori*.	Refined schema can inherit characteristics of a higher level and modify for next phase.
Large Data Volume	Thousands of sub-components in a circuit.	Limited only by physical storage; however, record-oriented storage may limit the size of record, causing multiple record retrievals for a single object.	Clustering by object reduces the number of disk accesses. Complex data types remove object size restrictions.
Long Transaction Duration	Designer takes two weeks to modify a specific circuit design.	Built around short business transactions. Inefficiency and failure occur with long transactions.	More appropriate concurrency control and failure recovery methods used to support long transactions.
Fast Performance	Thousands of sub-components are retrieved and displayed in seconds.	A single view requires multiple joins and many individual accesses.	Designed to retrieve large amounts of data at once.

Table 1.1 DBMS Support of Engineering Design Tool Characteristics (10)

There are currently no commercially available ODMSs that include a complete API for the Ada programming language. This creates a major problem for the Department of Defense (DoD) whose standard programming language is Ada. DoD projects are becoming increasingly complex and data intensive. With this complexity, the need for ODM continues to increase. One example can be found in Wright Laboratory's Functionally Integrated Resource Manager (FIRM) program. The FIRM program will provide a real-time database management system for the integrated avionics systems that will be a part of future fighter aircraft. The Ada programming language will be used in this effort along with an object-oriented database management model for avionics data management (13). In order for DoD applications, such as the FIRM program, to take full advantage of object-oriented data management, an Ada API must be developed.

This problem is complicated by the lack of standardization in ODM. Most relational DBMS vendors supply capabilities which at least match those defined in the ANSI SQL standard. This is nice for defining a generic interface for any relational DBMS, but there is currently no SQL-like standard for ODM systems.

1.3 Approach

Hedstrom (8) identifies two possible approaches to integrating Ada (or any other programming language) to an ODM system architecture. The first type of approach is called *loosely coupled.* A loosely coupled interface does not assume the data model of Ada necessarily matches the data model of the database. This approach acknowledges the need to interface Ada to existing databases, but the obvious disadvantage is that data translation is required at the application interface since data types used in database systems do not always match Ada data types. One example of a loosely coupled interface to Ada is the one provided by ORACLE, a popular relational system. The second type of approach is called *tightly coupled.* A tightly coupled interface is one where the Ada data model is equivalent to the object oriented data model. Such an interface would be *seamless* meaning that persistent data and transient data would be handled identically. A seamless interface makes interfacing with the database easier. In addition, Ada types would match database types eliminating the need for data conversion. The problem with a tightly coupled Ada interface is that Ada extensions may be required to support objects. In addition, since there is not a single object model, a unique extension for each ODM system may be required. The existence of Classic Ada with persistence provides evidence that a tightly coupled solution can be found. Features of Ada 9X may also prove helpful in realizing a tightly coupled interface (8).

The primary focus of this research effort is on the design of a loosely coupled Ada interface. The goal is to define an interface that is portable to different ODM systems. The Air Force Institute of Technology (AFIT) currently owns three ODM systems: Itasca, ObjectStore, and Matisse. AFIT plans to use these systems for research projects and for educational purposes. This study investigates the possibility of a single Ada interface package for all three of these systems. This would enable Ada programs to access any of the three ODM systems through the use of one software package. If possible, such an interface would prove extremely valuable, not only to AFIT, but to a number of DoD programs developing Ada-based software.

The approach used in this effort was to derive a set of requirements based upon current literature in ODM, features of existing commercial systems, and the needs of new applications. An interface was then designed to realize these requirements trading off the goals of portability, transparency, functional completeness, and performance. The design takes advantage of Ada's language interface capabilities to make calls to the existing C APIs which all of the ODM systems have in common.

1.4 Materials and Equipment

Among the equipment used in this research effort was a Sun Sparc Station II, a SunAda compiler, the ObjectStore DBMS, the Itasca DBMS, and the Matisse DBMS. The Ada ODM interface was designed within this environment. All of these items are available in AFIT laboratories.

1.5 Document Summary

In the next chapter we look at what features are expected in an ODM system and present an overview of the systems used at AFIT. In addition, Ada's object-oriented capabilities and limitations are investigated. In the third chapter, we present the requirements for an Ada ODM interface and a design solution based on a set of design goals. In the fourth chapter, we explain the techniques that can be used to implement the design and then look at the implementation challenges such a design presents. In the final chapter, we draw conclusions based on this research and present recommendations for future work.

II. Literature Review

2.1 Overview

The ultimate goal of this effort is to provide the benefits of persistent storage and ODM to Ada application developers. What is persistent storage and why is it important? What features characterize an ODM system? In this chapter we answer these questions by reviewing current literature in ODM. In addition, we provide an overview of ODM systems used at AFIT.

2.2 Persistence

Persistence is the concept of preserving data in a reusable state. From the programming language point of view, persistence is the ability of the programmer to have data survive the execution of a program, in order to reuse it in another program. Persistence can be very useful to a software developer. The persistence model for an ODM system can be *language centered* or *language neutral.* Language centered means that the persistent data model is based on an individual programming language. The close correspondence between the data model of the ODM system and the data model of the programming language results in less of a seam between persistent and transient data. Language neutral means that the ODM system has its own data model unlike any particular programming language. This approach facilitates sharing of objects between languages (18).

Persistence should be transparent to the developer. Typically, however, the structure of data within a programming language is quite different than its structure for long-term storage. Programming languages have many different ways to structure data efficiently for a particular application. Programmers can create and manipulate trees, lists, graphs, sets, and more. Data that is required to persist (long-term storage) must then be stored in the flat file system traditionally provided by operating systems. This dichotomy between short-term program structures and long-term file structures causes a considerable amount of program code to be used for conversion of data structures (14). Typically, 30% of program code is concerned with transferring data to and from files or a DBMS (2). It would obviously be very beneficial from a programmer's point of view to have all of the

I/O and data structure conversions done automatically. The amount of code could be cut considerably, thus increasing productivity and enhancing program maintenance.

Persistence is listed in (3) as a mandatory feature for ODM systems. The authors note that persistence in an ODM should be *orthogonal*. This means that each object, independent of its type, should be allowed to be persistent without explicit translation by the programmer. Attempts have been made to make a persistent Ada through various research prototypes and also through a commercial product known as Classic Ada with persistence. Classic Ada, a product marketed by Software Productivity Solutions, Inc., extends standard Ada with object-oriented constructs. It does so by providing object-oriented constructs which are reduced by a preprocessor to legal Ada constructs. Classic Ada with persistence provides an extra keyword, **persistent**, so that a user-defined class can be declared persistent. The primary disadvantage of this product is that all instances of a persistent class must be stored and retrieved as persistent objects. In other words, all instances must be persistent or none are persistent. Therefore, the orthogonality requirement is not satisfied.

2.3 ODMS Features

The task of designing an Ada ODMS is complicated by the lack of standardization in the field. In recent years, a consensus has emerged on what constitutes a minimal ODM system. Two important papers were written in 1989 by proponents of both the extended-relational and the OOPL approaches. The first paper, "The Object-Oriented Database System Manifesto", cites a number of mandatory rules that a database system must satisfy to be classified as an ODM system. The second paper, "Third-Generation Data Base System Manifesto", describes three general tenets for ODM systems and then describes necessary features to support the tenets. The authors argue that the industry should focus on integrating relational technology with ODM concepts. While both papers differ on implementation approach, a close look reveals broad agreement on the ultimate results (11). In (4), Barry compiled a list of possible features for an ODM system using the two manifesto papers and eighteen other sources. This list is used to organize the following discussion of ODM features.

2.3.1 Object Model.

Support for the object-oriented programming paradigm is what truly distinguishes ODM systems from traditional DBMSs. ODM systems manipulate *objects* which represent not just persistent data (attributes) but also operations on the data. Object-oriented principles include data abstraction, encapsulation, classes, inheritance, and object identification. Data abstraction is the ability to define abstract data structures composed of a variety of data types. The data structures could be trees, sets, graphs, queues and so on. Using these structures, objects can be grouped together and treated as one entity. Data abstraction make it easier to model real-world complexity for which relational systems are inadequate.

Encapsulation is a key concept in object-oriented technology. The concept states that objects should hide their code and data, while providing an external interface for operations on the data. Thus, programs that manipulate the objects are only concerned with the behavior of the operations on the objects and not the implementation of operations. This allows the implementation to change without disrupting programs that use the data. Encapsulation helps to reduce the cost and time required to produce complex applications by making it easier to maintain and improve code (12).

Another way to reduce programming costs is through inheritance. Inheritance is the ability to derive new classes from existing classes. Objects that share the same attributes and behavior are grouped together in a class. A class is analogous to an abstract data type but may also be a base type such as an integer or string. Classes are organized into class hierarchies where they can have subclasses or superclasses. A subclass inherits the attributes and behavior of its parent class. Inheritance provides support for reusable code and data which saves development costs.

Object identification is another important object-oriented principle. Many ODM systems automatically supply unique object identifiers (OIDs) for each persistent object. There are advantages to using OIDs. OIDs are immutable and completely independent of changes in data values or physical location. Also, OIDs provide a uniform way to reference objects.

2.3.2 Transaction Properties. A transaction is a program unit that reads and possibly writes various data items. Transactions are used as tools to support the traditional database functions of concurrency and recovery. This is because transactions must satisfy the *atomicity*, *consistency*, *isolation*, and *durability* (ACID) properties. Atomicity means that all operations in the program unit are completed or none are completed. Ideally, atomicity should be maintained in the presence of deadlock, database software failures, application software failures, CPU failures, or disk failures. Consistency means that a transaction provides a transistion between consistent states. Isolation means that information does not flow between active tansactions. In other words, a transaction's results are not revealed to other concurrent transactions until its results are committed. Durability means that the effects of a transaction are permanent in the database. This permanence should be maintained in the face of the various types of failures mentioned above.

The transaction is a well-established DBMS concept, but the field of ODM has introduced the concept of long duration transactions. A long duration transaction may last hours or days. For instance, in a Computer Aided Design (CAD) system, a design engineer may need to work on a particular design over the span of several days. In the meantime, one of his fellow engineers may wish to work on the design concurrently. This could mean major delays for the second engineer if the design is locked for several days. Another problem with long transactions is the possibility of system shutdown or failure. One way of handling long duration transactions is to implement a version management facility. A version is like a snapshot of a design object at some phase of the design process. An object can range in complexity from the primitive object which models a simple real-world entity to complicated composite objects used to model more complex design entities. By creating different versions of objects, concurrent design work can be performed on the same design and then the versions can be merged later. Version management is an essential feature for ODM systems designed for CAD applications (1).

Another important type of transaction for both the traditional DBMS and the ODM system is the nested transaction. Nesting transactions allows a finer grain of control for rolling back persistent state. In addition, allowing nested transactions enables a routine that initiates a transaction to be called from within a separate transaction (12).

2.3.3 Locking and Concurrency Control. Locking and concurrency control techniques are used to manage multiple users interacting concurrently with a DBMS. With respect to concurrency control, ODM systems should provide the same level of service currently provided by second generation systems. Concurrency control can be *optimistic* or *pessimistic*. Optimistic control assumes that simultaneous access to the same object is rare and therefore only checks conflicts at commit time. Pessimistic control usually involves locking mechanisms where access conflicts are checked when the locks are requested. Optimistic control techniques provide better performance if conflicts are indeed rare. Otherwise, optimistic control could suffer from many aborted transactions.

Locking techniques are an important feature of ODM systems. Locking can occur at the class or instance level. A complex composite object may be locked as a design entity or locks may be allowed on individual objects that make up the composite object. Locks may be escalated from instance level to class level when a large number of instances are locked in order to improve performance. Automatic promotion of locks from shared to exclusive may occur based on the activity involved. Many of these locking features are transparent to the user, but some control may be allowed and therefore locking could be a consideration when designing an interface.

2.3.4 Schema Modification. Frequent schema changes are necessary in many of the complex applications for which ODM systems are designed. Schema modification implies modifying programs that use the old data schema, modifying existing instances of the modified types, and considering effects of the changes on the remainder of the schema. The overhead associated with schema changes can be eased by maintaining data independence. This is the ability to change data stored in a database (such as database definition or schema) with minimum impact on existing applications. As described above, encapsulation of procedures is one way to achieve this goal. One example where this is particularly important is in design applications. Users of design applications are just as likely to change schema information as the application programmer. For example, a designer of some electronics system may need to define a new electronic component or define a design constraint (6).

2.3.5 Query Capability. One of the most important features of a database system is the ability to retrieve and manipulate data. Query languages have proven to be indispensable in relational systems, and most ODM researchers are in agreement about the importance of providing some form of associative access to data in ODM systems as well (3, 15). This is in contrast to the navigational point of view which is allowed by some systems. Navigational access to data implies the ability to access a specific record using a low-level procedural interface. One or more elements of the record would be a pointer to another record. The application could then navigate through by dereferencing pointers to establish new current records. The authors of (15) argue against this type of access for a number of reasons mostly due to the effect on schema evolution. If indexes on data change or data is reorganized by clustering, the programmer must change his program to account for the modified physical access paths. This compromises data independence which adversly affects program maintenance and development costs. Access to data using a non-procedural query language provides a mechanism for data independence.

2.4 Overview of AFIT ODMSs

As mentioned in chapter one, AFIT owns three ODM systems: Objectstore, Itasca, and Mattisse. These systems are extended object-oriented database programming languages (OOPLs). Extended OOPLs differ from the other categories in their use of the object-oriented data model and in their integration with existing programming languages. Extended OOPLs are designed to provide a unified programming interface for both persistent and transient data. This close integration of programming language and data model helps to overcome the impedance mismatch problem between applications and the database model. The database system enhances the programming language by providing concurrency control, persistence, a query language, and other traditional DBMS capabilities. Despite being classified similarly, the AFIT ODM systems differ in a number of significant ways. The following is an overview of the systems used at AFIT.

2.4.1 Objectstore. Objectstore, developed by Object Design of Burlington, Massachusetts, is an ODM system designed to make C++ a database programming language.

This means Objectstore can be classified as language centered since the data model is based on an individual programming language. Any type of C++ or C data can be made persistent including C++ objects. Persistent data stored using Objectstore is manipulated by C++ or C programs in the same way as transient data. The capability to declare data persistent is provided by overloading the memory allocation operator for C++ and C.

Objectstore provides a library of collection types including sets, lists, bags, and arrays. Collections are a convenient means of storing and manipulating groups of objects. By using different types of collections, the user obtains a great deal of control over the behavior and representation of groups. For example, collections can be ordered or unordered, and they can allow duplicates or prohibit duplicates. Collections can be used to model one-to-many and many-to-many relationships and they provide a convenient domain for executing queries (12).

Short-term concurrency and recovery is handled in Objectstore through a conventional transaction model using two-phase locking. Long-term concurrency and recovery is provided using a version control facility. Objects can be grouped into design entities known as *configurations*. Configurations can be checked out into private *workspaces* where they can be modified and then checked back into shared workspaces. Configurations can be locked while checked out to prevent simultaneous updates or alternate versions can be used to allow concurrent design work.

Objectstore can be used for the development of either C or C++ applications. There are four approaches to using Objectstore: the C library interface, the C++ library interface without class templates, the C++ library interface with class templates, and the C++ library interface with class templates and the Objectstore database manipulation language (DML).

2.4.2 Itasca. Itasca, developed by Itasca Systems of Minneapolis, Minnesota, uses object-oriented principles and is classified by Cattell as an extended OOPL due to its close association with Lisp (6). However, the intention of Itasca developers is to provide language neutral access to Itasca. Itasca neutrality is compromised somewhat by its dependence on Lisp. For example, query expression construction is dependent on Lisp syntax.

An application program does not need to be written in an object-oriented language to make full use of Itasca functionality since Itasca itself encapsulates a complete object model. Objects in Itasca have a unique identifier along with a state and behavior. The state of each object is represented by a set of attributes. The behavior is defined by a set of methods (code). Objects that share the same set of attributes and methods are grouped together in a class object. Itasca supports inheritance by allowing subclasses to be derived from existing classes. The resulting schema is a class hierarchy. Persistence is automatic for Itasca objects.

Itasca has all the features one might expect from any DBMS including concurrency control, transaction management, multiple security levels and recovery. Itasca also features functions to perform queries on database objects. Several features distinguish Itasca from others. Itasca supports dynamic schema modification. Users with security access can make changes to the schema at any time without affecting other parts of the database at the time the changes occur. Long duration transactions are supported by allowing users to check objects out of the shared database into their own private databases. Users can then modify the checked out objects without affecting the shared database or other users. The changes can then be checked back into the shared database or committed to the private database.

Itasca is a distributed database. The shared partition is distributed across sites in a network. Itasca clients provide transparent access to all parts of the shared database. Any number of private databases can exist at each distributed site (9).

2.4.3 Matisse. Matisse, developed by Intellitic International, is the third ODM system used at AFIT. Matisse uses its own data model and is not tied to any one programming language. Matisse provides all the traditional database features, but comes up short in certain areas specific to ODM. It doesn't support long transactions and has a limited versioning facility. One of the biggest drawbacks for interfacing purposes is that Matisse doesn't provide a query capability from an API. Queries must be performed from a graphical browser tool or must be hand-coded.

2.5 Object-Oriented Progamming in Ada

One of the principle advantages of the object-oriented programming paradigm is that it encourages building and extending software systems from reusable parts. The Ada language supports reusability through its package facility which is the mechanism through which abstract data types (ADTs) can be defined. ADTs encapsulate data and behavior and are analogous to classes. This has prompted some to classify Ada as an object-oriented language (5). However, Ada does not have some essential features necessary for directly implementing object-oriented programming such as inheritance of classes and dynamic binding of messages. Correcting these deficiencies is among the goals of Ada 9X (8).

2.6 The Ada/Objectstore Prototype

Object Design, Inc, produced a prototype Ada interface in January of 1992. This prototype was the basis for Li Chou's work at AFIT to improve the accessibility of Objectstore to Ada programmers. The Objectstore prototype provided functions and procedures for manipulating databases, transaction management, and persistent declarations. Chou added access to the collection facility. The Ada/Objectstore interface was accomplished by using Ada pragma interface statements to call C library routines from within Ada applications. The work of Object Design and Li Chou has shown that these Ada bindings provide access to the functionality of Objectstore without significant degradation in performance. Since the Ada bindings are the basis for the design described in this document, the technique is discussed in chapter four.

2.7 Summary

The field of ODM is new and still developing. While no standards have yet been defined, common features are evolving from commercial systems and research prototypes. This chapter presented a summary of features that characterize ODM systems and looked at some specific systems used at AFIT. Some of the features are standard for database management and can be found in traditional relational systems while others are more unique to ODM. In the next chapter, we define requirements for an Ada ODM interface and propose a design.

III. Requirements and Design

3.1 Overview

The purpose of this chapter is to identify the requirements for an Ada ODM interface and to present an Ada package specification for the interface. Design goals are also examined. The requirements are derived from the OODBMS manifesto (3), Barry's feature list (4), discussions with Major Roth, and studies of the three AFIT ODM systems (Objectstore, Itasca, and Matisse). The following requirements specification only describes *what* is required for the interface and not *how* it will be implemented.

3.2 General Requirements

The following section presents the general requirements for ODM using Ada. The requirements are based on the needs of Ada programmers and not on the capabilities or limitations of any one system. Ideally, ODM requirements should be evaluated from the perspective of the intended application, but the focus here is on what aspects of persistence and ODM functionality need to be visible to all Ada programmers, and on what needs are shared by a large set of applications. This excludes ODM features which are particular to a given system or application. In addition, this section ignores features over which the ordinary user exercises no control such as the locking mechanism.

3.2.1 Initialization. Most systems require some type of initialization process. This could be a means of collecting security information such as a user name and password, or the system may require parameters to indicate server or client information. Itasca, for example, requires a connection to the Itasca image to be established by calling **Iconnect_itasca** before using any C API functions. In order to handle any required initialization, the Ada interface should include some appropriate utilities.

3.2.2 Persistence. The basic requirement for persistence is that Ada programmers have a means of declaring data persistent. Also, persistence should be transparent so that the programmer need not worry about how the data maps between memory and storage. In addition to these requirements, there are other aspects of persistence which are desirable but not required. One issue concerns the question of what types can be declared persistent. As discussed in the previous chapter, data type orthogonality is a desirable feature. Ada programmers need to be able to declare defined Ada types as persistent objects as well as any user-defined types. Another important feature is persistence independence. This means that the persistence of a data object is independent of how the program manipulates that data object. For example, an application should be able to call a procedure with either persistent or transient objects as parameters (2). The degree of orthogonality and independence is dependent on the choice of the persistence model: language centered or language neutral.

Support for persistent data which can be shared by a number of processes concurrently implies the need for access control and some type of transaction mechanism. Also, the way in which data is referenced and manipulated will effect the syntax for persistence support. For these reasons, the functions and procedures for supporting the persistence requirements appear in the following sections.

3.2.3 Database Access. ODM systems will typically maintain a database using some operating system file. This file contains a schema description as well as the data. Application programmers require a means of manipulating the database files. In addition, access control is usually required to protect the data from misuse. To support these requirements, the interface should include facilities to create, destroy, open, and close database files.

3.2.4 Transactions. All access to persistent data must take place within a transaction. This ensures the information seen by applications is in a consistent state. The transaction model can be presented to applications in various ways. A transaction model can be flat or nested. As noted in chapter two, nested transactions are a desirable feature and should be implemented if possible. Another issue is whether or not the start and end of transactions are explicitly exposed to applications. For example, these events could be implicit in the beginning and end of a program. Unrecoverable errors would then be treated as a call for an abort. By having an explicit start transaction call, the user can define a nesting structure in a straight forward manner. Also, concurrency is improved if transactions are only used when necessary. For these reasons, the transaction model should include a start transaction call, a commit transaction call, and an abort call.

3.2.5 Object Requirements. Cattell defines objects as real-world or abstract entities that we model in a database (6). Object implementation can take many forms. An object can be a simple integer or an Ada record, however, ODM applications must deal with complex objects that may consist of multiple data types, may have relationships to other objects, and may be associated with specific behavior (code). Ada programmers need a way to represent and manipulate complex objects in their ODM applications.

Data representation in a database is described by the database schema. Users of the Ada ODM interface must have a consistent method of specifying the schema regardless of the system being used. This includes the different classes of objects that will be used in the application. In addition, the interface should provide the ability to create and delete objects. In order to manipulate objects, users will need to refer to objects through the use of unique object identifiers, also known as OIDs. Many ODM systems automatically generate unique identifiers. Some systems allow direct access to these system-generated OIDs to refer to objects.

A data value held by an object has a name and a type. These data values, or attributes, can take on a simple or a complex value. Simple attributes are those that take on literal values such as integers and strings. Complex attributes are those that define a set or a reference to another object. They are used to represent one-to-one, many-to-one, and many-to-many relationships. Ada programmers will need operations in the interface to specify initial attribute values, update these values, and get the values.

Encapsulation of data and code is another important requirement for objects. The user should be able to define operations for a class of objects. Ideally, these operations should then be stored in the database along with the data for the class. By storing the procedures in the database, the implementation of the procedures can be modified without affecting any existing application programs.

3.2.6 Query Capability. Ada ODM users need a simple way to query data. Most ODM systems feature a facility for expressing a query over some collection of objects. As a minimum, the query function will take a class name and a query string as parameters. Return values are typically some set that satisfies the query criteria. In addition, many systems feature a query function that returns a single element. This is more efficient when it is known that only one object element is sought. The Ada ODM interface should include analogous operations.

3.2.7 Schema Evolution. Cattell (6) identifies three important classifications for schema modifications. Considering the object requirements identified above, all three classifications are important for defining schema evolution requirements for the Ada ODM system interface. The first category involves changes to the components of a class. The user must be able to add an attribute, drop an attribute, change the name of an attribute, or change the type of an attribute. Similarly, a user should be able to add or drop a method, change the name of a method, or change the implementation of a method. The second category involves changes to the class hierarchy. This would involve adding a new supertype or subtype relationship or removing an existing supertype or subtype relationship. The final category concerns changes to the classes themselves. The user might need to add a new class, drop an existing class, or change the name of an existing class.

3.2.8 Long Duration Transactions. Current commercial ODM systems have generally addressed the problem of long term concurrency control in two ways. The first is through the use of *conversational transactions* which allow data to be checked out for a long period. The second solution is through the use of multiple object versions which allow coordination among multiple users. Ada users need a way to handle long transactions, but the semantics between current ODM system implementations vary widely and no standard set of features has evolved. This creates a problem in defining requirements for the Ada ODM interface since some systems have more flexible versioning capabilities than others. This problem is addressed again in the next chapter.

3.3 Design Goals

The overall goal is to provide persistence and ODM functionality for Ada programmers. The approach, as outlined in chapter one, is to create an Ada API to existing ODM systems. Keeping this approach in mind, their are certain high level requirements that are desirable. These requirements are called design *goals* since they may not be 100% realizable. Instead, they are considered to be desirable characteristics of an ideal interface and are kept in mind throughout the design process as ultimate goals.

3.3.1 Portability. A worthwhile goal of any software development effort should be portability. This means that the system is flexible enough to suport changes in hardware, external software, and commercial off-the-shelf (COTS) products with minimal effect on design and implementation (17). In the case of an Ada ODMS interface, portability means the interface can be used with any underlying ODM system. This goal is achieved by only including capabilities in the interface that are common to all ODM systems. This is not a simple task since no standard currently exists which ensures a minimum set of capabilities for commercial ODM products. The requirements outlined in the previous section serve as the basis for the ensuing analysis and design.

3.3.2 Transparency. Another important characteristic for an Ada ODMS interface is transparency. This involves the hiding of information from the user related to implementation. The Ada programmer should not need any knowledge of another programming language or of a particular ODM system. The programmer should be able to write a complete Ada application using regular Ada function or procedure calls to access the ODM facilities without special constructs.

3.3.3 Completeness. The Ada programmer should have access to all of the functionality for the ODM system being used. This would seem to be in conflict with the transparency goal since different ODM systems have different features and, therefore, may not map very well into a transparent interface. In other words, the user may require knowledge of the underlying ODM system when using certain features. In addition, the support an ODM system offers for a feature can vary considerably. The variation in features such as version management makes it tempting to take a least common denominator approach and only include capabilities common to all systems. This approach is undesirable since valuable database functionality is lost, but may be unavoidable in some cases. The tradeoffs between transparency and completeness have a significant effect on design decisions.

3.3.4 Performance. A final goal concerns performance. The Ada ODMS interface should not significantly degrade the performance of the ODM system or systems. This is an implementation issue and is discussed further in the next chapter.

3.4 Design

The design of the Ada ODMS interface can best be described as a layered approach. Figure 3.1 shows the overall concept. Each application communicates to the interface package directly, although in certain cases a preprocessor is necessary to provide a measure of transparency during application development. This is discussed further below. The interface layer, which is implemented in Ada, communicates with the API of the vendor specific ODM system. The ODM system alone is responsible for the transfer of data back and forth from non-volatile storage.

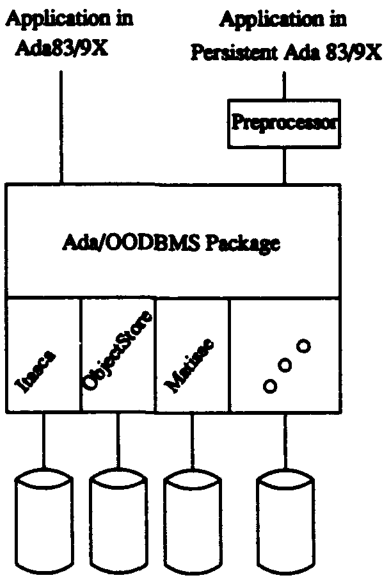

Figure 3.1 Overall Design

Figure 3.2 shows the breakdown of the different components of the interface which consists of two layers. The application interface layer is visible to the application and contains the main interface package. This package includes all of the core functions and procedures which must be supported by all ODM systems which use the interface. In addition to the core functionality, some systems may provide very useful features which are not included as part of the main interface package. To support this possibility, one or more annex packages can be added to handle functionality that is only supported by a subset of vendors.

The first layer serves to insulate the application from the details of the database interface layer. The database interface layer contains the bindings which allow Ada programs to use code written in another language. This method of binding Ada to the API of the ODM system is discussed in detail in the next chapter. Due to these bindings, the database interface layer is specific to a given system. However, if the ODM system is to be changed, this is the only layer that would be affected. Any such changes are transparent to the application. This layer may consist of several packages that together make up the interface to a certain system.

3.4.1 The Preprocessor.

The persistent Ada ODM interface makes use of a preprocessor. There are several reasons why this might be useful. Up to this point, the discussion has been limited to general requirements and design without considering specific ODM systems. However, in order to understand the need for a preprocessor, the Objectstore application development process must be understood. The Ada ODMS interface is designed to make the application development process simple and straightforward. The Ada programmer writes an ODM application completely in Ada. ODM functionality is accessed by simple procedure and function calls to the interface layer. Ideally, everything from initial schema definition to querying existing data can be accomplished in this manner. The Objectstore ODM system presents some roadblocks to realizing this concept. Objectstore application developers must have a separate schema source file. The schema source file specifies all classes and types in the application that are read from or written to persistent storage. By comparison, schema definition in Itasca can be accomplished

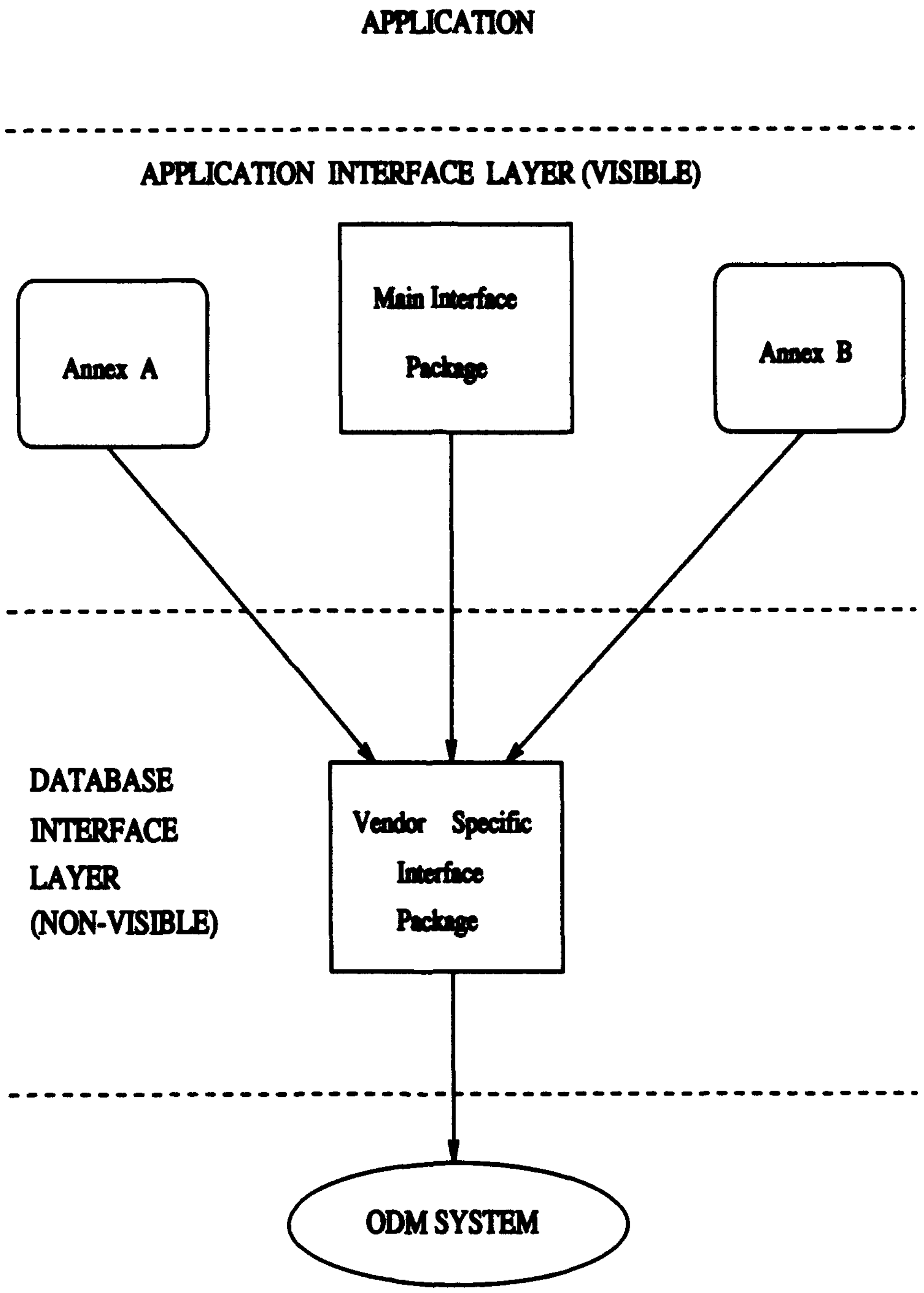
APPLICATION
APPLICATION INTERFACE LAYER (VISIBLE)
Annex A
Main Interface Package
Annex B
DATABASE INTERFACE LAYER (NON-VISIBLE)
Vendor Specific Interface Package
ODM SYSTEM

Figure 3.2 Breakdown of Interface Layers

within the application. A preprocessor would allow a resolution of the two different application development processes. Special constructs can be used by the Ada programmer to define a schema within an application. The preprocessor would then scan the application to determine what is required, and then create a database and set up the schema for the underlying ODM system.

A preprocessor could also help in situations where the syntax used in the API of an ODM system is tied heavily to a certain programming language. This means the Ada programmer must know something about the other language to use the interface. Objectstore and Itasca both have examples of this in their respective C APIs: Objectstore queries are expressed using a predicate written in C, and Itasca queries are expressed using a predicate written in LISP. A preprocessor would allow the queries to be expressed in Ada and converted automatically to the proper form.

3.4.2 Schema Specification File. The application development process, using the Ada ODM interface, is a one or two step process depending on what must be done. If a new application will be using an existing database with a schema already defined, then development consists of simply writing the application using the functionality available in the main interface package and any annexes. If the database does not exist, the developer must prepare a separate schema specification file along with the main application.

The schema specification file is used by the preprocessor to create the desired database and set up the initial schema required by the application. A database is an operating system file. The developer specifies the name of the new database file and all of the data types that will be used in a persistent context. If inheritance is supported by the ODM system, the class hierarchy is also specified at this point using special constructs. The preprocessor then can read the schema specification file, create the database, and set up the schema. If Objectstore was being used, this would mean creating the schema source file which would be used to run a simple program to create the database. The schema source file would then be used with subsequent applications that use the database. If Itasca was the underlying system, this would mean just running a program to do all the work since the C API contains all the functions needed to create a database and schema. The reasoning behind

Table 3.1 Main Interface Package Operations

Requirement	Operation
Initialization	INIT_ADA_ODM STOP_ADA_ODM
Database Manipulation	DB_OPEN DB_CLOSE DB_DESTROY
Object Manipulation	MAKE_OBJECT DELETE_OBJECT UPDATE_ATTRIBUTE GET_ATTRIBUTE ADD_TO_SET REMOVE_FROM_SET
Transaction Management	START_TXN COMMIT_TXN ABORT_TXN
Queries	QUERY QUERY_PICK

a schema specification file was driven by the desire to resolve two very different ways of doing the initial schema creation. Specifically, the Objectstore way and the Itasca way. However, this concept is general enough to use with any system.

3.5 Operation Summary

Table 3.1 categorizes one possible set of operations for the main interface package. This is a minimal set based on the requirements outlined above. Any ODM system would certainly be expected to support the operations listed in the table. The semantics of these operations are described in the next chapter. Appendix A contains the complete Ada specifications. There are no operations for schema definition or database creation since this is accomplished by the preprocessor. Instead, a syntax is needed so that the user can specify the schema and database name in the schema specification file. Ideally, this syntax would look a lot like Ada. The next chapter deals with mapping the functionality from the table to specific ODM systems used at AFIT. Additional operations are also discussed along with possible annex packages.

Operations to support schema evolution and version management are not included because approaches and semantics between ODM systems vary so widely. In fact, these capabilities are better suited as annexes to the main interface package. These important requirements are addressed in the next chapter when specific system implementations are discussed.

3.6 Summary

This chapter specified requirements for an Ada ODM interface. Design goals for the interface were then outlined. These goals include portability, transparency, completeness, and performance. In addition, a design was proposed to meet the requirements and goals. Finally, a minimal set of operations was presented for the main interface package. In the next chapter, we investigate the issues involved in implementing these operations.

IV. Implementation Issues

4.1 Overview

The purpose of this chapter is to describe how the Ada ODM interface would be implemented using the ODM systems at AFIT and how the development process would work. Itasca and Objectstore are used in the descriptions since they provide the largest contrast in approaches. There is not just one correct way to implement each required function. The implementation of the database interface layer can be accomplished in different ways. However, the semantics of each function should be the same regardless of the ODM system being used.

The implementation described in this chapter is based on the use of the C API for both Itasca and Objectstore. It is also based on the use of Ada bindings to the respective C APIs. The following sections discuss the Ada binding technique and some type considerations. The remainder of the chapter deals with the mapping of Itasca and Objectstore functionality to that required by the interface.

4.2 Sun Ada Interfacing

The Ada programming language includes facilities to make calls to programs written in a different programming language. This section describes the way Ada programs can make calls to C routines using the Sun Ada development environment. This technique is used to enable the database interface layer to communicate with the ODM system being used.

4.2.1 Ada Bindings. Sun Ada communicates with other programming languages through the use of *pragmas* which are instructions to the compiler. The pragma statements INTERFACE and INTERFACE_NAME allow Ada programs to call subroutines defined in C, Ada, PASCAL, and FORTRAN. The technique is best illustrated by the following example:

```
procedure c_start_objectstore;
pragma INTERFACE(C, c_start_objectstore);
pragma INTERFACE_NAME(c_start_objectstore,
                      "start_objectstore");
```

```
5:     procedure START_OBJECTSTORE is
6:     begin
7:        c_start_objectstore;
8:     end START_OBJECTSTORE;
9:     pragma INLINE(START_OBJECTSTORE);
```

This code fragment was taken from the Ada/Objectstore prototype. It appears in the body of the interface package and is used to call the Objectstore C API function `start_objectstore` which performs some required initialization of the system. The specification for the package contains the declaration for the Ada procedure `START_OBJECTSTORE` which is all that is visible for the programmer's use. Line 1 is a declaration of the C routine that will be called, line 2 indicates to the compiler that the procedure `c_start_objectstore` uses a C language protocol, and lines 3 and 4 inform the compiler of the actual library name the routine should be mapped to at link time. The C routine is then called within the Ada procedure `START_OBJECTSTORE` on lines 5 through 8. The final line uses the pragma `INLINE` which causes the Ada compiler to expand the Ada routine inline where called, rather than generating call/return instructions. This results in reduced stack manipulation and fewer subprogram calls during execution.

4.2.2 Data Type Representation. The biggest problem with designing the bindings is ensuring that any Ada types used in conjunction with the C subroutines are of a compatible data representation in both languages. The Objectstore function used in the example above has no parameters. However, many functions used by Objectstore and Itasca do require parameters, and if the Ada program does not use a parameter with the word size, value range, or alignment expected by the C routine, erroneous results could occur.

The Sun Ada programmer's guide (16) cites two approaches to creating parallel data representations. The first approach is to simply read the vendor's documentation and use types that are known to be parallel between the two languages. For example, the type `INTEGER` in Ada corresponds to the type `int` in C, while the type `SHORT_INTEGER` is equivalent to the type `short` in C. Problems can arise with this approach due to the fact that neither Ada nor C compilers are required to use a particular size to represent any particular type. Therefore, an implementation is free to choose a representation based on hardware considerations. The second approach is to use Ada representation clauses which allow the Ada programmer to define an exact duplicate of the physical layout of any data type in another language once it is known. These type definitions are then largely independent of the implementation so that type, record layout, and alignment can be controlled. For example, a C unsigned integer is one type that does not have a predefined Ada equivalent. This type can be created using representation clauses as follows:

```
type C_UNSIGNED_SHORT is range 0 .. (2 ** 16) - 1;
for C_UNSIGNED_SHORT'LENGTH use 16;

u_var : C_UNSIGNED_SHORT;
```

The first statement creates an Ada type with a range of values equivalent to the C type's range. The second statement guarantees that at most 16 bits of storage are allocated to every object of this type, and then the final statement declares a variable of the new type.

One thing to keep in mind when matching C types to Ada types is that the pragma interface statements only permit 32-bit or 64-bit scalar values to be passed. As a result of this limitation, Ada types such as SHORT_INTEGER must be passed to a C function using an address. Fortunately, system address is a predefined Ada attribute defined in the SYSTEM package and can be used for this purpose. Compound types such as arrays or records can also be passed using a system address since both Ada and C associate the label of compound types with a base address. As long as the compound types are composed of equivalent base types, the structure of the compound types will be identical including the offsets for accessing the individual components.

A final consideration for data representation between Ada and C is string types. A C string is terminated by a null character and is represented by a pointer to the first character. An Ada string is represented by a pointer to an unconstrained array of characters and has a length attribute. Some provision must be made when passing Ada strings to C routines to make the appropriate conversions. For a complete mapping of Ada and C types, see (7).

4.3 Mapping Functionality

The main interface package contains the standard functions and procedures that must be supported by the underlying ODM system. This section describes how Itasca and Objectstore can be used to implement these facilities. The main goal here is to present one interface to the application regardless of whether the Itasca interface package or the Objectstore interface package is used in the database interface layer. Each section features an explanation of how the procedure or function from the main interface package is used followed by a discussion of the implementation issues.

4.3.1 Initialization. The Ada ODM interface includes two procedures to signal the beginning and end of access to the ODM system. `INIT_ADA_ODM` needs to be called at the beginning of any Ada application which uses the interface. Both Itasca and Objectstore require an initial function to be called at the beginning of any application which uses their respective C APIs. The Itasca function `Iconnect_itasca` requires four parameters while the Objectstore function `start_objectstore` has no parameters. To resolve this difference, the `INIT_ADA_ODM` procedure can have default values for the Itasca parameters. This way, if Objectstore is used, the parameters can be omitted when calling `INIT_ADA_ODM`. Another more general option is to use Ada's overloading abilities. The interface could include an overloaded version of `INIT_ADA_ODM` for each ODM system that could be used. Using default values is the recommended approach since transparency is a major goal.

`STOP_ADA_ODM` is the final function called at the end of any Ada application which uses the interface. Itasca requires the function `Idisconnect_itasca` to be called at the end of each C application to close the connection to the Itasca image. There is no analogous Objectstore function. Therefore, a call to `STOP_ADA_ODM` causes `Idisconnect_itasca` to be called in the Itasca package, but is a no-op in the Objectstore package.

4.3.2 Database Manipulation. There are three procedures in the main interface package for manipulating databases: `DB_OPEN`, `DB_CLOSE`, and `DB_DESTROY`. Creation of databases is done automatically by the preprocessor. The Ada ODM interface uses a conceptual view of databases similar to Objectstore. In Objectstore, a database is a separate operating system file. Access to databases is controlled by setting permissions (upon creation) much like one would do with a UNIX file using the chmod command. Itasca's view of databases is a little different. Itasca has one database file at each distributed site. Each of these files has one shared database partition and multiple private partitions. Users don't make changes to the shared partition directly. Changes to data are made by checking out objects from the shared, distributed database, making the changes in a private database, and then committing the changes to the shared database at any time later. Functions in the C API can be used to control access to the private databases.

Itasca's conceptual view was intended to support design applications where a team of engineers work on one project. By using the shared and private database concept, work can be accomplished by multiple engineers simultaneously on different parts of a design. Objectstore supports design applications through its version management facility. Central to this facility is the concept of a *workspace* and a *configuration.* A configuration serves to group together objects that are treated as one design. A workspace provides context for both shared and private work. A global workspace contains one or more configurations and can be accessed by all users. Each user can then create their own workspace as a child of the global workspace for their own private use. Configurations can be checked out from the global workspace to the user workspace and can be checked in to the global workspace from the user workspace.

It is easy to see how Objectstore's version management facility can be used to simulate the Itasca conceptual view. For example, a single database with a global workspace could be created for each design project. A `DB_CREATE` procedure could be added to the interface which would create a user workspace as a child of the global workspace. This would be analogous to an Itasca private database. The functionality in the version management facility could then be used to map to Itasca functions. For example, the version management facilty has check-in and check-out functions similar to those of Itasca. The problem with this approach is that it uses Objectstore's robust version management facility in a very limited way. In addition, this is an awkward approach for applications that don't really need the separate shared and private workspaces. By using Objectstore's view of databases, the version management facility can be added as an annex. In Itasca, a private database will correspond to a database file. The three procedures in the main interface package would all operate on private databases. The shared database would not be used. This approach uses Itasca in a restrictive way, but it is favored because of its generality.

It is useful to maintain the concept of a current database as Itasca does. In Itasca, all work is done in the context of the current private database. In Objectstore, the database must be specified as a parameter in many functions since multiple database files can be open. By allowing only one database file open as the current database, the user is freed from including the database as a parameter with every function that requires it. The Objectstore interface can keep up with the database name and pass it to the functions. Also, the database parameter would be meaningless to the Itasca interface since it is not needed once the current private database has been set. While this approach is recommended, there is an associated cost. Objectstore users will lose the ability to open and manipulate multiple database files simultaneously in the same application.

4.3.3 Transactions. Control over transactions is handled through three procedures. `START_TXN` is used to signal the beginning of a transaction *`COMMIT_TXN` commits* all changes to persistent data since the last commit, and `ABORT_TXN` aborts all changes to persistent data since the last commit. These procedures can all be directly implemented by simply making calls to the analogous functions in either Objectstore or Itasca. Itasca doesn't have a function to explicitly start a transaction, but this can be simulated by calling the Itasca function `Icommit` whenever `START_TXN` is called.

As long as a flat transaction model is used, the simple implementation above works well. Implementing nested transactions is a little more complicated. Objectstore allows nested transactions and provides additional functions to support this concept. Each call to the Objectstore function `transaction_begin`, returns a pointer to a transaction. Normally, this pointer is then assigned to a variable within the application. The pointer is then used to call the functions `transaction_commit` and `transaction_abort`. The transaction pointers are also used as parameters in the functions `get_current` and `get_parent` to allow the user to reference a transaction at any nesting level. Keeping track of pointers could be handled transparently by the Objectstore interface package, but then the ability to abort or commit a transaction at any nesting level is lost.

Itasca's documentation doesn't explicitly advertise support for nested transactions, but has a mechanism for doing so. A session in Itasca is a program unit which encapsulates a sequence of transactions. The `Iopen_session` function starts a session and the `Iclose_session` function ends a session. Sessions are activated and deactivated in a stack-like fashion. When `Iopen_session` is called, the current session is suspended. After a call t› `Iclose_session`, the most recently deactivated session is reactivated. Therefore, to simulate nested transactions, each call to the procedure `START_TXN` can open a session, and each call to `COMMIT_TXN` or `ABORT_TXN` can close a session.

It is probably best to sacrifice Objectstore's ability to reference any nesting level in the interest of transparency. The user will then have to abort or commit transactions starting from the innermost level. In addition, the Objectstore interface will need to keep up with the transaction pointers.

4.3.4 Initial Schema Definition. The preprocessor is responsible for creating databases and for creating the initial schema definition for each database. It does this using information provided by the application developer in the schema specification file. Creating applications in this way ensures that the developer will have a consistent, transparent development process regardless of which ODM system is being used.

In Objectstore's Ada interface prototype, Ada objects are directly represented in the Objectstore database by a corresponding C type. Since any C type can be made persistent in Objectstore, any Ada type which has a corresponding C representation can be made persistent. In Itasca, persistence is obtained by defining the attributes and operations for a class of objects. All user-defined classes are derived from an Itasca root class. Persistence is automatic. The attributes of the classes are made up of C types.

The Ada record type is used to represent an object in the Ada ODM interface. If Objectstore is used, the record will correspond to a C structure. If Itasca is used, the record will correspond to an instance of an Itasca class. Itasca automatically groups all instances of a class together for query purposes. This concept is useful in Objectstore as well. Queries in Objectstore are performed on collections which are groups of objects. To support this in the Ada ODM interface and to appear semantically identical to the Itasca implementation, all newly created objects in Objectstore can be placed in a collection representing their class. This can be done at the database interface layer and thus be transparent to the application developer. This should also be optional for each Objectstore class due to the overhead involved. The developer could specify in the schema specification file whether objects should be grouped into a collection.

The application developer specifies classes in the schema specification file similarly to a normal Ada record declaration. No special keywords are necessary because the record's declaration in the schema specification file implies the use of a persistent class. In addition, the class hierarchy must be specified. Superclasses and subclasses are designated using constructs similar to those proposed by Ada 9X. Itasca allows this type of class structure to be formed, but direct implementation of inheritance is not possible in Objectstore using the C API. This is because the C programming language does not support inheritance. The implementation of inheritance in Objectstore is dependent on the use of C++ constructs.

One solution to this problem is to have the Objectstore preprocessor generate a database and schema using C++. The subsequent storage and access of data would then be done using function calls from the Ada ODM main interface package. These functions would use the C API to manipulate data. The C++ API is not used for implementing the main interface package because the C API is easier to match to Ada. Most functions in the C library can be exactly replicated in Ada with return values and parameters lists being the same. This is not always the case in the C++ API. For example, the Objectstore function `create_root` features a database name parameter in the C API version. However, the parameter is not needed in the C++ API since `create_root` is a member function of the class `database` (7).

Databases are designated for creation by the keyword `DB_CREATE` which is consistent with the database operation syntax used in the main interface package. The name of the new database is specified along with the `DB_CREATE` command. This name is used to refer to the database at the application level. A full path name is used as a parameter in Objectstore functions to refer to databases. Itasca assigns a number to each newly created database. The user must use that number to reference the database in subsequent operations. To resolve this difference, the preprocessor can create a database configuration file which maps the database name at the application level with the Objectstore pathname or the Itasca number. Each ODM interface can then reference this file as necessary. The following code illustrates how a simple schema specification file might look:

```
DB_CREATE new.DB

TYPE Person IS RECORD
   name : string;
   ssn : string;
END RECORD;

TYPE Student IS NEW PERSON WITH
   gpa : float;
   major : string;
END RECORD;

TYPE Employee IS NEW PERSON WITH
   salary : float;
   supervisor : string;
END RECORD;
```

Ada syntax is retained as much as possible for the convenience of the Ada programmer. The `Student` and `Employee` class are subclasses of `Person`. They will inherit the attributes of `Person` in addition to their own attributes. See Appendix B for the complete syntax.

If Objectstore is being used, the preprocessor creates a schema source file that must be used with subsequent applications, and a UNIX file that serves as the database. For Itasca, a private database partition is created with the specified schema.

4.3.5 Object Manipulation. Three operations are provided in the main interface package for manipulating database objects. `MAKE_OBJECT` creates an instance of a class. Parameters include the class name and the values of any attributes. `MAKE_OBJECT` corresponds to the Itasca function `Imake`. To implement `MAKE_OBJECT` in the Objectstore interface, an Ada record corresponding to the given class name must be persistently allocated using the given attribute values. This record object must then be added to the collection representing the class. This can all be done at the database interface layer so that MAKE_OBJECT is semantically identical using either Itasca or Objectstore.

The DELETE_OBJECT operation removes an object from a class. It corresponds to the Itasca function Idelete_object. With Objectstore, the object must be removed from the appropriate collection and then deleted from storage. Again, this can be accomplished in the Objectstore interface so the application need only call DELETE_OBJECT.

The UPDATE_OBJECT operation is used to change an attribute value for a given object. Parameters include the OID, the attribute name, and the new value. The analog Itasca function is Iupdate_attribute. Objectstore allows changes to persistent objects directly rather than through function calls. For instance, the fields of a persistent C structure can be changed in the same way as its transient counterpart. There are certain advantages to requiring that all assignments in the Objectstore C API be accomplished through calls to UPDATE_OBJECT. To understand these advantages, it helps to understand how Objectstore supports relationships. In C++, 1-to-1 relationships are represented by an embedded class with only a single pointer field. For multi-way relationships, the embedded class contains an Objectstore collection. Assignment and initialization are overloaded to effect proper relationship maintenance. This type of relationship maintenance is not supported in the Objectstore C API. Support of relationships can only be achieved the same way in C by requiring that all assignments be accomplished through function calls since the assignment operator is primitive and not overloadable. By requiring the use of UPDATE_OBJECT to change an object attribute value, relationship maintenance can be handled on the database interface level. The drawback is the associated loss in persistence transparency. Persistent Ada records (objects) require special manipulation not required by transient Ada records.

The MAKE_OBJECT operation must be implemented as an Ada function since it returns an OID for the new object. This OID can then be used as a parameter in the DELETE_OBJECT or UPDATE_OBJECT operation. Both Objectstore and Itasca use pointers to pass OIDs. These pointers can be mapped to suitable Ada integer types to use in the Ada interface.

One major drawback of implementing objects in this way concerns methods. There is no way to encapsulate code with individual objects and still maintain an acceptable level of transparency. Itasca features the ability to define methods for a single object, but the code is written in Lisp. In addition, this ability does not map to a similar ability in Objectstore. Objectstore associates code with objects through the abilities of the C++ programming language. Normally, Ada programmers will associate code with abstract data types using Ada's package facilities. However, this won't work in the implementation described above since the objects are not actual Ada types. The Ada record is used to define the class, but the actual object representation in the database is a C structure (Objectstore) or an Itasca class instance. Manipulation of the objects is accomplished through predefined functions. These objects can't be declared as a type in an Ada package.

4.3.6 Queries. The commands to perform queries are `QUERY` and `QUERY_PICK`. These represent two basic commands that any ODM system should be able to support. `QUERY` takes a class name and returns a set of objects that satisfies a given query expression. `QUERY_PICK` returns a single object that satisfies the query. Both of these operations have corresponding functions in Itasca. Objectstore also supports these two functions directly, but the Objectstore query functions must be used with collections.

The problem with implementing queries lies with the query expression parameter used in each function. Itasca uses a Lisp expression while Objectstore uses a C expression. Either a common syntax should be used at the application level and translated into an appropriate expression, or the application developer must be famaliar with the particular ODM system's way of expressing queries. Requiring the developer to be famaliar with Lisp or C would be in conflict with the goal of transparency so creating a common syntax would be preferred.

4.4 Annexes

The main interface package contains a basic set of operations that any system must support. However, there are some ODM concepts that are very important but are more difficult to standardize. This is because the concepts are realized in ODM systems in a variety of ways with varying degrees of sophistication. By allowing optional annexes as part of an Ada ODM interface, we can suggest a standard way of implementing an ODM concept, based on a specific system, while realizing that not every system can conform. The following sections suggest two possible annexes for Objectstore and Itasca.

4.4.1 Schema Evolution. Itasca features a fairly sophisticated schema evolution capability. All of the schema evolution requirements discussed in the previous chapter are supported by Itasca. A similar schema evolution ability could be proposed for the Ada ODM interface since the object models are similar. The functional mapping would be very similar since an Ada record corresponds to an Itasca class. Abilities would include adding or deleting record fields, changing the field types, adding or deleting record classes, and changing the class hierarchy. All of this can be done in Itasca by simple calls to the appropriate functions. Most importantly, schema changes in Itasca can be done dynamically while an application is running.

Objectstore does not feature the kind of dynamic schema evolution supported by Itasca. Changes to the schema must be defined before compilation in a separate schema source file. Other than this problem, Objectstore allows similar changes to the schema. Mapping schema evolution for these two systems into a common, transparent process would require some preprocessing just like the initial schema definition. This preprocessing would mean Itasca users would lose the ability to implement schema changes dynam ically. To avoid limitations on Itasca's schema evolution abilities, an annex could be provided for these functions.

4.4.2 Version Management. Both Itasca and Objectstore feature version management but the two interpretations are quite different. As explained earlier, Objectstore allows objects to be grouped together in configurations which belong to a particular workspace. The workspaces form a parent/child hierarchy of arbitrary depth. This arrangement allows many degrees of sharing and privacy. Workspaces higher in the hierarchy have greater degrees of sharing than lower ones, and workspaces lower in the hierarchy feature greater degrees of privacy. Versions are created by checking out (or locking) a configuration from a shared workspace to a private one.

In Itasca, there are four levels of versions for an object: transient, working, released, and generic. A transient version can be updated or deleted by the user who created it. Also, new transient versions can be derived from existing ones. The existing ones are then promoted to a working version. Working versions are considered stable and cannot be updated, but can be deleted by their owner. Promotion to a released version can be user-specified or system-determined. A released version cannot be updated or deleted. Generic versions are used as general object references. For instance, if a generic object is deleted, the entire version-derivation hierarchy is deleted.

Itasca controls sharing of information through its concept of a shared database and multiple private databases, however, this conceptual view is lost in the Ada ODM interface since the Objectstore view of individual database files is being used. In other words, only Itasca private databases are being used in the Ada ODM interface. Rather than limiting Objectstore's version management to simulating Itasca's four version levels, the Objectstore version management facility could be introduced as a separate annex.

4.5 Summary

This chapter has taken a look at the challenges involved in implementing the Ada ODM interface. The binding techniques discussed are a proven method of interfacing Ada to an ODM system. The most significant example is the Ada/Objectstore prototype developed by Object Design and extended by Li Chou (7). The functional mapping described in this chapter represents a mix of concepts from both systems in an attempt to support a basic set of operations for Ada application developers. Annexes are provided to support desirable concepts which can't be supported by all ODM systems.

The implementation of the main interface package operations can be accomplished in different ways. The final chapter looks at the conclusions that can be drawn from this research and recommends future directions for research in this area.

V. Conclusions and Recommendations

5.1 Overview

This chapter takes a look at the advantages and disadvantages of the design approach and the degree to which the design satisfies the design goals. In addition, recommendations for future work in this area are presented.

5.2 Conclusions

In this thesis, we defined a set of requirements and proposed a design for an Ada ODM interface. The design goals were portability, transparency, and completeness for Ada programmers. Portability means that the ODM system can be changed without affecting any existing application programs. Transparency means that Ada programmers can use the interface without having to know different programming languages or specifics about the different systems being used. Completeness means that all of the functionality of the ODM systems are available to Ada application developers. All of these goals were considered equally desirable during the design process. While planning the implementation using Itasca and Objectstore, it became apparent that the goals could not be completely realized. Lack of standardization in ODM systems was the major obstacle in reaching these goals. In (17), Voketaitis describes a portable RDBMS interface for Ada. The success of this effort was largely due to the standardization of relational systems through SQL. By creating an interface that utilizes only ANSI-SQL compliant features, functional compatibility with other vendor products is ensured. This is not the case with ODM systems. There is no standardization in the field of ODM which makes defining a common Ada interface more difficult. A standard needs to be defined, but existing systems may have difficulty conforming. Implementation of a prototype for Itasca and Objectstore should be guided by a prioritization of the design goals. If portability is paramount, reduced capability is a likely consequence. If functional completeness is the concern, a direct interface to the one system that best meets an application's requirements might be a better solution. The following sections discuss the merits and drawbacks for the proposed interface.

5.2.1 The Preprocessor. The preprocessor concept was employed to help meet the goal of transparency during application development. Objectstore requires that a new database schema be specified before compilation in a schema source file. Itasca allows programmers to set up a new database schema within the application. The Ada ODM interface requires new database schemas to be specified in a schema specification file. This file is then read by the preprocessor which creates the database and its associated schema. Itasca users lose the ability to create databases and schemas at run time. This could be a disadvantage in some applications. Some other disadvantages of using preprocessors include the cost of development, testing, and maintenance.

5.2.2 Loosely Coupled versus Tightly Coupled. Many of the problems associated with implementing the Ada interface are due to the fact that the object models and the data types differ between Ada and the ODM systems and between the ODM systems themselves. This would prompt some to suggest that a tightly coupled interface would be the answer for ODM with Ada. However, this argument ignores the need to interface Ada to existing commercial systems. Past experience with relational systems indicates that, over time, large repositories of legacy data will be established. Ada applications will require access to this data just as they do to relational databases today.

5.2.3 Interfacing to a Single System. Having established the need to interface to existing systems, some may argue that a direct interface to one system would be more effective. With one system, the Ada interface only has to conform to one object model. Also, fewer compromises would have to be made in functionality. The problem again lies with the fact that no dominant product has emerged and no standards have yet been defined. Dependence on one product could be disastrous if that product fails to succeed in the marketplace. A portable interface will allow development to proceed despite the state of flux in the industry.

5.2.4 Language Neutral vs Language Centered. The proposed implementation in chapter four is based on an object model similar to Itasca's language neutral model. This was done to facilitate support of relationships and sharing of objects. In addition, inheritance can be provided to Ada programmers by using Itasca's model. The problem with the language neutral data model is choosing a data model. The data model of the Ada ODM interface represents a least common denominator approach which gives up power. A more complicated data model would make it more difficult for some ODM systems to conform.

Another approach would have been to try and make the Ada ODM interface language centered like Objectstore. Using Objectstore's philosophy, any Ada type could be made persistent and would be manipulated the same way as transient data. This would have been easy to do using Objectstore since it was designed with seamlessness in mind, but in Itasca, some manipulation would be required to make the interface appear seamless to the Ada programmer. At issue here is whether or not the goal is a persistent Ada or just ODM access for Ada.

5.2.5 Itasca and Objectstore. Chapter four described how the operations of the Ada ODM interface could be implemented using Itasca and Objectstore. This was a good exercise to point out the challenges associated with implementing two ODM systems with very different approaches to ODM. The result is a mix of concepts from both systems. For instance, Itasca's object model is used along with Objectstore's database file concept. Some implementation decisions were based on necessity: Itasca could not support referencing nested transactions. Other decisions were made based on practicality: simulating Itasca's shared and private database using Objectstore's version management facility would be awkward and limiting. The remainder of this section looks at some drawbacks of the proposed implementation.

5.2.5.1 Inheritance. Inheritance is not available when using Objectstore with the Ada ODM interface. This is because inheritance is implemented in Objectstore through constructs associated with C++ classes. This is a good example of a limitation of the language centered approach. Objectstore was designed to extend the C++ language and thus using other languages can result in reduced capability. A solution to this is to have the Objectstore preprocessor set up databases and schemas using the C++ API and then use the C API to implement the object manipulation functions in the main interface package. By contrast, Itasca can support inheritance regardless of the language being used. Itasca has objects that are not particular to any language. Class hierarchies are set up through function calls. Since inheritance is a requirement for the Ada ODM interface, the C++ API should be used for initial schema specification in the Objectstore preprocessor.

5.2.5.2 Persistence. In Objectstore, any C++ or C type can be made persistent. Objectstore was designed so that transient and persistent data could be manipulated the same way. Itasca classes are automatically persistent, and there is a clear distinction in an application program between transient and persistent data. The Ada ODM interface follows Itasca's persistence model. This could be viewed as a disadvantage for Ada programmers. A seamless interface (such as Objectstore with C++) provides a more computationally complete interface. Ada programmers would certainly benefit from the ability to designate any type persistent and subsequently use the persistent types in normal computations.

5.2.5.3 Operations. One important concept of the object-oriented paradigm is the ability to encapsulate code and data into a single object. Itasca allows the definition of Lisp routines that can be associated with an object and stored in the database. Objectstore associates code with objects through C++ classes, but does not store that code in the database. Both of these concepts are problematic for the Ada ODM interface. Requiring Ada users to know Lisp would conflict with the goal of transparency. On the other hand, since the C API is being used in Objectstore, the ability to associate code with objects through C++ classes is lost.

Normally, Ada programmers associate code with data through Ada's package facility creating an Abstract Data Type. This couldn't be done in the proposed implementation since a persistent object is not a specific Ada type. Although classes are defined using Ada record constructs in the schema specification file, no record declarations are necessary in the main application. Objects are handled this way to prevent users from trying to manipulate objects as if they were Ada records.

5.3 Recommendations for Future Research

Implementing a prototype interface would help to further isolate problem areas. However, this could get to be quite an effort. Creating the bindings to the respective C APIs would be relatively simple, but a preprocessor must be developed for both Itasca and Objectstore. The person or persons attempting such an effort must be proficient in Ada, C, Objectstore, and Itasca. Other potential areas for further research and development include:

- *Relationships* - As explained in chapter four, Objectstore does not provide support for relationships in the C API. However, this support could be added in the Ada ODM interface since updates to object attributes are accomplished through function calls. These functions could be used to maintain referential integrity. This should be studied further to determine the best implementation.
- *Queries* - Itasca queries are expressed using Lisp syntax while Objectstore uses C syntax. This is a common problem among the different vendors. It would be useful to establish a common syntax to express queries for Ada programmers. The interface could then translate this syntax into an appropriate expression for the respective ODM systems. The preprocessor could also be made responsible for the translation.
- *Ada 9X* - Ada 9X includes object-oriented enhancements including support for inheritance. The impact of Ada 9X on the Ada ODM interface should be investigated. Ada 9X may make interfacing to the C++ API in Objectstore more feasible. This would allow the interface to be more functionally complete since Objectstore was designed for C++.
- *SQL3* - One of the main difficulties in defining an interface that is portable to multiple ODM systems is the lack of standardization. Implementation of many ODM concepts varies considerably between systems. The new SQL standard has potential to improve this situation by adding ODM concepts. One area of study could be to investigate the potential impact of the new SQL standard on the Ada ODM interface.

5.4 Final Comments

As the Ada programming language becomes more widely used for information systems development, a viable interface between Ada and ODM systems is a critical requirement for success. This thesis is part of an on-going effort at AFIT to develop a portable, Ada ODM interface. Li Chou's work showed that a loosely coupled interface using Ada bindings to the C API of an existing ODM system is possible without serious degradation in performance. This effort defined requirements for a portable interface. A design was proposed including Ada specifications for a minimal set of functionality. The groundwork is now set for implementation of a prototype interface.

Appendix A. Ada Specifications

This appendix contains the Ada specifications for the main interface package and the proposed annex packages.

A.1 Main Interface Specification

The following Ada Specification includes the functions and procedures of the main interface package. The specification takes advantage of some Ada features such as default parameters and overloading. For example, the INIT_ADA_ODM procedure features some default parameters since Itasca requires a user name and password for its initialization function while Objectstore does not. The procedures used for attribute manipulation are overloaded since different data types are involved. Some of the required overloadings are shown.

```
with SYSTEM; use SYSTEM;

package ADA_ODM is

  type TRANSACTION is private;
  type OBJECT is private;
  type SET_TYPE is private;

----------------------------------------------------------------------
  procedure INIT_ADA_ODM(USER: STRING:= "none";
                         PASSWD: STRING:= "none";
                         SERVER: STRING:= "hawkeye");

  procedure STOP_ADA_ODM;
----------------------------------------------------------------------
  procedure OPEN_DB(DBNAME: STRING);

  procedure CLOSE_DB(DBNAME: STRING);

  procedure DESTROY_DB(DBNAME: STRING);
----------------------------------------------------------------------
  function START_TXN return TRANSACTION;

  procedure COMMIT_TXN(TXN: TRANSACTION);

  procedure ABORT_TXN(TXN: TRANSACTION);
```

```
---------------------------------------------------------------------
  function MAKE_OBJECT(CLASS: STRING; ATTRIBUTES: STRING) return OBJECT;

  procedure DELETE_OBJECT(OBJID: OBJECT);

  procedure UPDATE_ATTRIBUTE(OBJID:OBJECT; ATTR:STRING; VALUE:OBJECT);
  procedure UPDATE_ATTRIBUTE(OBJID:OBJECT; ATTR:STRING; VALUE:INTEGER);
  procedure UPDATE_ATTRIBUTE(OBJID:OBJECT; ATTR:STRING; VALUE:FLOAT);
  procedure UPDATE_ATTRIBUTE(OBJID:OBJECT; ATTR:STRING; VALUE:STRING);
  procedure UPDATE_ATTRIBUTE(OBJID:OBJECT; ATTR:STRING; VALUE:CHAR);

  procedure GET_ATTRIBUTE(OBJID:OBJECT; ATTR:STRING; out VALUE:OBJECT);
  procedure GET_ATTRIBUTE(OBJID:OBJECT; ATTR:STRING; out VALUE:INTEGER);
  procedure GET_ATTRIBUTE(OBJID:OBJECT; ATTR:STRING; out VALUE:FLOAT);
  procedure GET_ATTRIBUTE(OBJID:OBJECT; ATTR:STRING; out VALUE:STRING);
  procedure GET_ATTRIBUTE(OBJID:OBJECT; ATTR:STRING; out VALUE:CHAR);

  procedure ADD_TO_SET(OBJID:OBJECT; ATTR:STRING; VALUE:OBJECT);
  procedure ADD_TO_SET(OBJID:OBJECT; ATTR:STRING; VALUE:INTEGER);
  procedure ADD_TO_SET(OBJID:OBJECT; ATTR:STRING; VALUE:FLOAT);
  procedure ADD_TO_SET(OBJID:OBJECT; ATTR:STRING; VALUE:STRING);
  procedure ADD_TO_SET(OBJID:OBJECT; ATTR:STRING; VALUE:CHAR);

  procedure REMOVE_FROM_SET(OBJID:OBJECT; ATTR:STRING; VALUE:OBJECT);
  procedure REMOVE_FROM_SET(OBJID:OBJECT; ATTR:STRING; VALUE:INTEGER);
  procedure REMOVE_FROM_SET(OBJID:OBJECT; ATTR:STRING; VALUE:FLOAT);
  procedure REMOVE_FROM_SET(OBJID:OBJECT; ATTR:STRING; VALUE:STRING);
  procedure REMOVE_FROM_SET(OBJID:OBJECT; ATTR:STRING; VALUE:CHAR);
---------------------------------------------------------------------
  function QUERY(CLASS:STRING; EXPRESSION:STRING) return SET_TYPE;
  function QUERY_PICK(CLASS:STRING; EXPRESSION:STRING) return OBJECT;
---------------------------------------------------------------------

  private
    type TRANSACTION is new INTEGER;
    type OBJECT is new INTEGER;
    type SET_TYPE is new INTEGER;

end ADA_ODM;
```

A.2 Annex: Schema Evolution

The following Ada specification includes the functions and procedures needed for schema evolution. This functionality has been placed in a separate annex since many systems cannot support dynamic schema evolution.

```
package SCHEMA_EVOLUTION is

  type OBJECT is private;

------------------------------------------------------------------------
  procedure ADD_CLASS(NAME:STRING; SUPERCLASS:STRING);

  procedure REMOVE_CLASS(NAME:STRING);

  procedure ADD_SUPERCLASS(NAME:STRING; SUPERCLASS:STRING);

  procedure REMOVE_SUPERCLASS(NAME:STRING; SUPERCLASS:STRING);

  procedure ADD_ATTR(CLASS:STRING; ATTR_NAME:STRING; INIT_VALUE:OBJECT);
  procedure ADD_ATTR(CLASS:STRING; ATTR_NAME:STRING; INIT_VALUE:INTEGER);
  procedure ADD_ATTR(CLASS:STRING; ATTR_NAME:STRING; INIT_VALUE:FLOAT);
  procedure ADD_ATTR(CLASS:STRING; ATTR_NAME:STRING; INIT_VALUE:STRING);
  procedure ADD_ATTR(CLASS:STRING; ATTR_NAME:STRING; INIT_VALUE:CHAR);

  procedure DELETE_ATTR(CLASS:STRING; ATTR_NAME:STRING);
------------------------------------------------------------------------
  private
     type OBJECT is new INTEGER;

end SCHEMA_EVOLUTION;
```

A.3 Annex: Version Management

Unlike dynamic schema evolution, many ODM systems do support version management in some form. However, the implementations can be quite different. Itasca's versioning relies on the shared and private database concept. Since the shared database is not used in the Ada ODM interface, some other versioning scheme is needed. This annex was added to support Objectstore's version management facility.

```
package VERSIONS is
```

```
    type CONFIGURATION is private;
    type WORKSPACE is private;

  ------------------------------------------------------------------------
    function NEW_CONFIG(PARENT:CONFIGURATION) return CONFIGURATION;

    procedure CHECKIN(ITEM:CONFIGURATION);

    procedure CHECKOUT(ITEM:CONFIGURATION);

    procedure DESTROY_VERSION(ITEM:CONFIGURATION);

    procedure DELETE_WORKSPACE(ITEM:WORKSPACE);

    procedure CREATE_GLOBAL_WS(NAME:STRING);

    function NEW_WORKSPACE(PARENT:WORKSPACE; NAME:STRING) return WORKSPACE;

    procedure SET_CURRENT_WS(ITEM:WORKSPACE);

    function GET_CURRENT_WS return WORKSPACE;

    function GET_PARENT_WS(ITEM:WORKSPACE) return WORKSPACE;

   ------------------------------------------------------------------------
    private
      type CONFIGURATION is new INTEGER;
      type WORKSPACE is new INTEGER;

  end VERSIONS;
```

Appendix B. Schema Specification File Syntax

This appendix describes the syntax used in the schema specification file. This file is prepared by the application developer to create a new database and database schema. The preprocessor is responsible for reading the file and performing the necessary operations. Subsequent applications can then populate the database, perform queries, or make changes to data values as needed.

B.1 Database Creation

Database creation is performed by the preprocessor. There is no separate function in the main interface package for creating a database as there is for opening, closing, or deleting a database. Specification is of the form:

```
DB_CREATE filename;
```

The keyword DB_CREATE is followed by a valid operating system file name. The schema for the new database is then specified below the DB_CREATE keyword. The developer may specify more than one database for creation. The keyword is used by the preprocessor as a delimiter to distinguish one database schema from another. Therefore, one complete database schema should be specified before a second one begins.

B.2 Schema Specification

A schema specification is a description of each class of objects that may be used in a database. This includes a class name, attribute names, attribute types, and inheritance hierarchies. Classes are specified much like a normal Ada record declaration. A class hierarchy can be designated by using syntax similar to the syntax used in Ada 9X to specify inheritance. A class declaration is of the form:

```
TYPE class1 IS RECORD

   attr1: type;
   attr2: type;
   attr3: type;
      .
```

```
          .
          .

END RECORD;
```

A subclass of `class1` could then be specified as:

```
TYPE subclass IS NEW class1 WITH

   attr4: type;
   attr5: type;
   attr6: type;
          .
          .
          .

END RECORD;
```

The class **subclass** would then inherit the attributes of `class1` along with its own attributes. Multiple inheritance is not allowed in the Ada ODM interface.

B.2.1 Attribute Domains. The record fields represent object attributes. Any discrete Ada data type can be used as an attribute value. The preprocessor will ensure an appropriate corresponding type is used in the actual database representation. In addition, attribute values can represent one-to-one, one-to-many, or many-to-many relationships. To support this, an attribute type can be a class name, a set of class names, or a set of some other discrete data type. Example syntax is shown in the last section below.

B.2.2 Class Extent Option. The Objectstore query functions operate on Objectstore collections. To support these functions, the developer must designate which classes will be represented by a collection. All new object instances will then be automatically grouped into a collection. Users designate a class extent by adding the keyword EXTENT to the normal class definition. For example:

```
TYPE EXTENT classname IS RECORD

   attr1: type;
   attr2: type;
          .
```

```
        .
        .
END RECORD;
```

Subclasses of an extent class do not automatically become extent classes. Extension must be separately indicated for each class.

B.3 Example

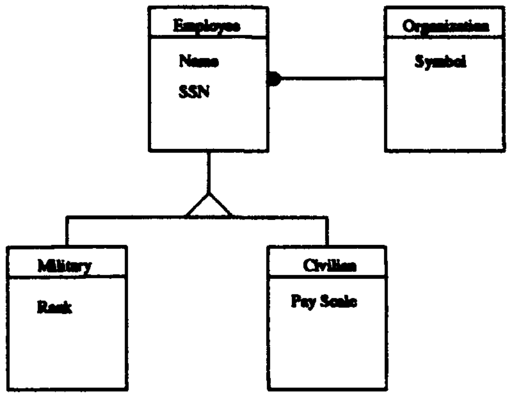

Figure B.1 Simple Object Model

The following example is provided to illustrate the concepts discussed above. Figure B.1 shows a simple object model with four classes, an inheritance hierarchy and a one-to-many relationship. In order to implement this model as a database schema, the following schema specification file would be used:

```
DB_CREATE employee.DB

TYPE employee IS RECORD
   name : STRING;
   ssn : INTEGER;
   office : organization INVERSE employees;
END RECORD;

TYPE Military IS NEW employee WITH
   rank : STRING;
END RECORD;

TYPE Civilian IS NEW employee WITH
   payscale : STRING;
```

```
END RECORD;

TYPE EXTENT organization IS RECORD
   symbol : STRING;
   employees : SET-OF employee INVERSE office;
END RECORD;
```

Inverse relationships are indicated using the keyword `INVERSE` as shown above. Queries can be performed on `Organization` since it was specified as a class extent.

Bibliography

1. Ahmed, Rafi and Shamkant B. Navathe. "Version Management of Composite Objects in CAD Databases," *SIGMOD Rec.*, *20*(2):218–227 (June 1991).
2. Atkinson, M. P., et al. "An Approach to Persistent Programming," *The Computer Journal*, *26*(4):360–365 (1983).
3. Atkinson, Malcom, et al. *The Object-Oriented Database System Manifesto*. Technical Report 30-89, LeChesnay, France: GIP ALTAIR, September 1989.
4. Barry, Douglas K. "ODBMS Feature List," *Object Magazine*, *2*(5):48–53 (February 1993).
5. Booch, G. *Object-Oriented Design with Applications*. Benjamin-Cummings, 1990.
6. Cattell, R.G.G. *Object Data Manangement*. Addison-Wesley Publishing Company, Inc., 1991.
7. Chou, Li. *Object-Oriented Database Access from Ada*. MS thesis, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, March 1993.
8. Hedstrom, J. *Taxonomy of Approaches to Interface Ada to an Object-Oriented Database*. Technical Report, Texas Instruments Incorporated, May 1991.
9. Itasca Systems, Inc., Minneapolis, MN. *Technical Summary* (2.2 Edition), 1993.
10. Jacobs, Captain Timothy M. *An Object-Oriented Database Implementation of The Magic VSLI Layout Design System*. MS thesis, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1991.
11. Melton, Jim. "SQL Update: The Moose is Loose," *Database Programming and Design*, *6*(5):70–73 (May 1993).
12. Object Design, Inc., Burlington, Massachusetts. *ObjectStore User Guide* (2.0 Edition), 1992.
13. Roth, Mark A., et al. "Database Management: The Heart of Integrated Avionics," *Proceedings IEEE NAECON*, 535–539 (1993).
14. Sajeev, A.S.M. and A.J. Hurst. "Programming Persistence in X," *IEEE Computer*, *25*(9):57–66 (September 1992).
15. Stonebraker, Michael, et al. *Third-Generation Database System Manifesto*. Memorandum UCB/ERL M90/28, College of Engineering, University of California, Berkeley, California 94720: Electronics Research Laboratory, April 1990.
16. Sun Microsystems, Mountain View, California. *Sun Ada Programmer's Guide* (1.1 Edition), 1992.
17. Voketaitis, A.M. "A Portable and Reusable RDBMS Interface for Ada," *ACM Ada Letters*, *12*(5):64–76 (September 1992).
18. Wells, David. *DARPA Open Object-Oriented Database Preliminary Module Specification: Persistence Policy Manager*. Technical Report, Texas Instruments Incorporated, November 1991.

Vita

Captain Anthony D. Moyers was born on 21 August 1964 at Barksdale AFB in Shreveport, Louisiana. He graduated from John Overton High School in Nashville, Tennessee in 1982 and attended Tennessee Technological University, graduating with a Bachelor of Science in Electrical Engineering in 1986. He then attended Officer Training School, receiving a commission as a Second Lieutenant in March 1987. Upon graduation, he was assigned to Robins AFB, Georgia, as an electronics engineer with the Airborne Electronics Division, Warner Robins Air Logistics Center. He entered the Graduate School of Engineering, Air Force Institute of Technology, in May 1992.

Permanent address: 8 Shoreline Drive
Fayetteville, Tennessee 37334

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE

AN OBJECT-ORIENTED DATABASE INTERFACE FOR ADA

5. FUNDING NUMBERS

6. AUTHOR(S)

Anthony D. Moyers, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/93D-11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Capt Rick Painter
2241 Avionics Circle, Suite 16
WL/AAWA-1 BLD 620
Wright-Patterson AFB, OH 45433-7765
(513) 255-4429 or DSN 785-4429

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Object data management (ODM) is currently at the forefront of research and development efforts in the database community. In order to take advantage of ODM technology, the Department of Defense (DoD) needs to ensure that these systems are accessible to Ada programmers. The Air Force Institute of Technology (AFIT) is conducting research towards the development of an Ada ODM interface to existing ODM systems. The design goals are portability, transparency, and completeness for Ada programmers. Portability means that the ODM system can be changed without affecting any existing application programs. Transparency means that Ada programmers can use the interface without having to know different programming languages or specifics about the different systems being used. Completeness means that all of the functionality of the ODM systems are available to Ada application developers. This thesis defines requirements for an Ada ODM interface and proposes a design. In addition, the challenges associated with implementation are investigated using commercial ODM systems at AFIT. Implementation of the interface is based on the use of Ada bindings to the existing application program interfaces (APIs) of the ODM systems. A preprocessor will be necessary in order to achieve transparency.

14. SUBJECT TERMS

ada; databases, object-oriented databases, bindings, interfaces

15. NUMBER OF PAGES 67

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL